Idea: State-Continuous Transfer of State in Protected-Module Architectures

Raoul Strackx and Niels Lambrigts

iMinds-DistriNet, University of Leuven,
raoul.strackx@cs.kuleuven.be
niels.lambrigts@gmail.com

Abstract. The ability to copy data effortlessly poses significant security issues in many applications; It is difficult to safely lend out music or e-books, virtual credits cannot be transferred between peers without contacting a central server or co-operation with other network nodes, ... Protecting digital copies is hard because of the huge software and hardware trusted computing base applications have to rely on. Protected-module architectures (PMAs) provide an interesting alternative by relying only on a minimal set of security primitives. Recently it has been proven that such platforms can provide strong security guarantees. However, transferring state of protected modules has, to the best of our knowledge, not yet been studied.
In this paper, we present a protocol to transfer protected modules from one machine to another state-continuously; From a high level point of view, only a *single instance* of the module exists that executes without interruption when it is transferred from one machine to another. In practice however an attacker may (i) crash the system at any point in time (i.e., a crash attack), (ii) present the system with a stale state (i.e., a rollback attack), or (iii) trick both machines to continue execution of the module (i.e., a forking attack). We also discuss use cases of such a system that go well beyond digital rights management.

Keywords: Protected Module Architecture, State-Continuity, Transferring State

1 Introduction

Computer science has transformed the world more rapidly than any technology before. Online encyclopedia enable worldwide access to knowledge, news travels faster than ever before and social media provide a global discussion platform. One key reason for this success is that data can be copied cheaply, easily and without loss of quality.

Unfortunately this ability also poses a security risk in many use cases. Digital rights management is the most obvious example: Once released, content providers cannot prevent that their data is distributed further. But many other use cases exist as well. We elaborate on valuable applications in Section 2.

Certain types of data should resemble physical objects; It should not be possible to create exact copies and once transferred to another party, the sender should no longer have access. This is hard to guarantee in practice. Security checks can be added to applications and operating systems, but commodity operating systems are so complex that their correctness cannot be guaranteed. A determined attacker is likely to find an exploitable vulnerability (e.g, buffer overflows [13,22]) in this huge trusted computing base (TCB). Alternatively, the owner itself could have an incentive to break the implemented security features and launch physical attacks against the machine (e.g., cold boot attacks) [4,5].

Protected-Module Architectures Recent advances in security architectures provide the required building blocks for an alternative approach. Protected-module architectures (PMAs) avoid a huge TCB by only providing a minimal set of security properties [6, 8, 10–12, 14, 20, 21, 24]. The exact set depends on the exact implementation, but all provide complete isolation of software modules. The PMA guarantees that modules have full control over their own memory regions; Any attempt to access memory locations belonging to the protected module at any privilege level (including from other modules), will be prevented. Protected modules can only be accessed through the interface that they expose explicitly.

Protected-module architectures can be used to harden security-sensitive parts of an application. Strackx et al. [20] evaluate a simple use case of a client connecting to a protected module. By placing SSL logic inside the protected module, only SSL packets cross the module's protection boundaries. Operating system services are still used to send and receive network packets, but these are *not* trusted. As any attempt to access sensitive memory locations in the protected module will be prevented by the PMA, this effectively reduces the power of a kernel-level attacker to that of a network-level attacker; Messages can be intercepted, modified, replayed or dropped, but sensitive data cannot be intercepted by an attacker.

Recently proposed protected-module architectures [6,14] also protect against sophisticated hardware attacks. Intel SGX, a PMA that is expected to be implemented in Intel processors in the near future, for example, guarantees that protected modules are only stored unencrypted when they reside in the CPU's cache. Before they are evicted to main memory or later to swap disk, they are confidentiality, integrity and version protected. This prevents many hardware attacks such as a cold boot attacks [5].

State-Continuity Guarantees Agten et al. [1,2] and Patrignani et al. [16,17] formally proved that protected-module architectures can guarantee strong security properties of modules *while they execute continuously*. In practice however machines crash, need to reboot or lose power at unexpected times. To deal with such events, modules must store their state on disk. However, confidentiality and integrity protecting module states before they are passed to the untrusted operating system for storage, is not sufficient. After the system reboots modules cannot distinguish between fresh and stale states. In many applications this forms a security vulnerability.

Parno et al. [15] and Strackx et al. [18, 19] propose a solution and guarantee state-continuous execution; Modules either (eventually) advance with the provided input or never advance at all.

Our contributions In this paper we build upon these security primitives and present a protocol to state-continuously transfer state of protected modules from one machine to another. At a high level, programmer's point of view, protected modules execute without interruption while they are transferred between machines. In practice however, we assume that an attacker may gain kernel- or hypervisor-level access to the system, but the underlying guarantees provided by the protected-module architecture cannot be broken. This implies that an attacker can (i) crash the system at any point in time, (ii) replay network messages, (iii) impersonate a remote party, and (iv) attempt to roll back the state of a module (e.g, by creating a new module and replay network packets). Since such a powerful attacker can easily launch denial-of-service attacks (e.g, by corrupting the kernel image), such attacks are not considered.

2 Use Cases

When it can be guaranteed that a protected-module instance cannot be rolled back nor forked when it is transferred to other machines, previously infeasible use cases become available.

Changing Machine Parno et al. [15] and Strackx et al. [18] proposed an algorithm to guarantee state-continuous execution of protected modules. While this property enables versatile use cases, it prevents protected modules to be passed to another machine. In practice however, machines need to be replaced. Our protocol provides such support. Note that backups do *not* provide a good use case as they imply that a module could be restored to a previous state. We explicitly wish to defend against such behavior.

Multi Processor-Package Systems & Cloud Computing Intel SGX provides strong security guarantees in face of software and hardware attackers by ensuring that protected modules are only stored unencrypted in the CPU's cache. When they are evicted to untrusted RAM or swap disk, they are confidentiality, integrity and version protected. Unfortunately, this also prevents protected modules to be transferred from one processor package to another on the same machine. A state-continuous enclave must always be executed on the same CPU package as it was created on. This poses significant practical challenges as applications can no longer be migrated from one CPU package to another to load balance execution load.

A similar problem occurs in a cloud computing setting where a virtual machine may be migrated from one physical server to another. State-continuous transfer of states can solve such problems.

Digital Wallets Cryptocurrencies such as Bitcoin continue to gain traction. Bitcoins can be transferred from one user to another, but network consensus must be reached before the transfer can be asserted. This has the disadvantage that bitcoins cannot be transferred when the sender cannot connect to the network.

By relying on security properties of protected-module architectures, an alternative infrastructure can be built easily. Virtual credits can be stored as a simple counter in a protected module, completely isolated from the rest of the system. Transferring credits between peers can then be achieved using a state-continuous transfer of a portion of the available credits. The protocol presented in section 3 can be trivially modified to enable such use cases (i.e., the sender no longer sends its current state, nor will it permanently disable itself after the protocol completed).

Distributed Capability Systems King-Lacroix et al. [7] propose BottleCap a capability-based mechanism for distributed networks. Resources can be accessed *iff* the user has the capability to do so. As access right checks are always local to the data that is accessed, such a system is much more scalable than traditional user-based authentication.

By placing capabilities in protected modules and transferring their state continuously, BottleCap's difficulties can be overcome; Capabilities can be easily revoked and special transfer policies for capabilities can be implemented easily.

Lending Digital Content Using the state-continuity properties we provide, digital content can be lent similar to physical objects; Once content is passed, it can no longer be accessed by the sender. A book, for example, could be lent out by state-continuously transferring the protected module it is stored in. When the user wishes to read a page, it is rendered inside the protected module and only the resulting image is passed to unprotected code. While significantly raising the bar for attackers, we acknowledge that this setup does not prevent attackers from copying the book by copying each rendered page. Additional technologies such as Intel IPT[1] can be used to mitigate such attacks.

3 Transferring State

The protocol that we present only relies on properties (that can be) provided by almost all protected-module architectures. Therefore, we will not assume any protected-module architecture in particular when presenting our protocol.

Say we wish to transfer the state of a protected module $M_{\texttt{src}}$ from one machine (i.e., `src`) to another (i.e., `dst`). Our protocol operates in two phases. In the first phase a new instance of module M with a "blank" state module is created on `dst` (called $M_{\texttt{dst}}$) and a public-private key pair is generated. When $M_{\texttt{src}}$ is guaranteed that the module was deployed correctly, it commits to the state transfer. This marks the beginning of the second phase where the state is transferred to $M_{\texttt{dst}}$. By encrypting the state with $M_{\texttt{dst}}$'s public key, *only* $M_{\texttt{dst}}$ is able to ever resume M's execution.

[1] `http://ipt.intel.com/`

The Protocol Figure 1 displays the protocol graphically in more detail. In the first step, $M_{\texttt{src}}$ generates a public-private key pair $(PK_{\texttt{src}}, SK_{\texttt{src}})$ and passes $PK_{\texttt{src}}$ to `dst` together with $M_{\texttt{blank}}$ (the module's code with a "blank" state) and a nonce n. Endpoint `dst` loads $M_{\texttt{blank}}$ in memory and starts its execution with $PK_{\texttt{src}}$ as argument. This public key will later be used to ensure that only states signed with the related private key will be accepted. The newly created module $M_{\texttt{dst}}$ also generates a public-private key pair $(PK_{\texttt{dst}}, SK_{\texttt{dst}})$ that will be used to transfer $M_{\texttt{src}}$'s state securely. The three resulting keys are all stored in $M_{\texttt{dst}}$ after which its correct execution is attested[2] to $M_{\texttt{src}}$ (step 4). The enclosed nonce ensures freshness.

After $M_{\texttt{src}}$ verified the attestation and found correct, it stops servicing user requests and uses the PMA's state-continuity property to commit to the state transfer; At some point in the future $M_{\texttt{src}}$ will transfer it's state to a module M with possession of $SK_{\texttt{dst}}$, or the module will never advance it's state again. In step 6 $M_{\texttt{src}}$'s state is encrypted with $PK_{\texttt{dst}}$ and signed with $SK_{\texttt{src}}$. It is passed together with a new nonce m to `dst`. After ensuring that the passed state originated from $M_{\texttt{src}}$, $M_{\texttt{dst}}$ decrypts the state and resumes the module's execution. To avoid that $M_{\texttt{src}}$ continues to attempt to transfer state, the protocols terminates by attesting that $M_{\texttt{dst}}$ accepted the state transfer. $M_{\texttt{src}}$ can now permanently disable requests to retry state transfer to $M_{\texttt{dst}}$ and destruct itself.

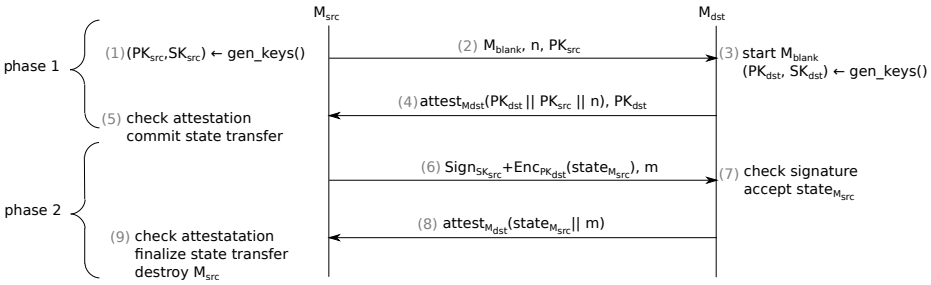

Fig. 1. Overview of the protocol where $M_{\texttt{src}}$'s state is transferred continuously to $M_{\texttt{dst}}$.

Security Analysis Safety of our protocol is based on two simple properties. First, we rely on the ability to create and store cryptographic keys in protected modules. At the beginning of phase 2 (i.e., after step 5), $M_{\texttt{src}}$ is committed to

[2] An attestation is a log of inputs and outputs that is signed by a trusted entity (e.g., a PMA platform or TPM chip). In a simple implementation a private key is embedded in the trusted entity that is signed by a trusted third party. This allows easy verification of the attestation log. Implementations such as Intel SGX [3] however use more complex cryptographic protocols to prevent linkability of attestation logs. We do not target a single PMA specifically in this section but simply state what needs to be attested.

transfer state from $M_{\texttt{src}}$ to $M_{\texttt{dst}}$. To ensure that only one module instance can ever resume execution of the module, the state is encrypted using the public key generated by $M_{\texttt{dst}}$. Correct implementation of $M_{\texttt{dst}}$ guarantees that the decryption key will never leak. Second, state-continuous execution of $M_{\texttt{src}}$ and $M_{\texttt{dst}}$ guarantees that neither module will ever be forked.

Loss of power or similar events during the execution of the protocol, may disrupt execution of the protocol. Such events during the first phase will cancel the state transfer. Attack events after the state transfer was committed however, will either (i) eventually result in another attempt to transfer the state to the *same remote module instance* (e.g., in case of power or network failure), or (ii) will lead to a situation where the module will never resume execution (e.g., machine `dst` is physically destroyed).

To ensure that the state originated from $M_{\texttt{src}}$, the protocol passes in step 2 $M_{\texttt{src}}$'s public key to $M_{\texttt{dst}}$. This additional security check is not strictly required, but omitting it leaves the protocol open to a denial-of-service attack from a remote attacker. As $M_{\texttt{dst}}$ then would then accept *any* provided state (and only accepts a new state once), a remote attacker could send it any state. In that case $M_{\texttt{src}}$ would never be able to complete its state transfer, but it also does not allow user requests to be handled until the state is transferred.

While guaranteeing that state can be transferred securely, the protocol does *not* provide any assurance about the origin of the module's state. An attacker masquerading as $M_{\texttt{src}}$ could even fabricate a state. In most cases this does not pose a vulnerability as it only needs to be guaranteed that a specific module instance will continue to execute continuously while it is transferred from host to host. For other use cases, a proof of the origin of the module can be easily added in two ways: (1) Verification of the origin of a module could be built in the module itself. When the module is initially created, for example, it could generate a secret shared with a verifier. Even when the module is transferred from host to host, possession of this secret proves its origin. Or (2) the protocol could be modified to also attest the correct set up of $M_{\texttt{src}}$ to $M_{\texttt{dst}}$ in step 2. This would prevent the fabrication of states by an attacker, but a verifier interacting with two module's over time cannot determine whether they are the same module instance.

4 Related Work

State-continuous transfer of state is easy when the user can rely on the correctness of the operating system. In practice this assumption is hard to guarantee. Protected-module architectures provide an interesting alternative, but to the best of our knowledge, state-continuous transfer has not been addressed in such a setting.

Related work that relies on a very limited TCB exist, but only provide very specific security guarantees. Van Dijk et al. [23], for example, present a system where a central server provide tamper-evident persistent storage. Rollback and forking attacks of this data is prevented using a trusted time stamping device

executing on an untrusted third party's server. Our protocol can achieve similar guarantees (i.e., clients can pass data with build-in protection against rollback or forking) but is more flexible. Modules for example, can easily choose to only transfer their state partially, as was discussed in the digital wallet example of Section 2. We also do not rely on a central server, which provides significant scalability advantages.

More recently Kotla [9] presented a way to build the digital equivalent of scratch-off cards. A client can choose to either use a cryptographic key stored in the TPM chip but cannot hide its use upon an audit. Or it does not access the cryptographic key and after this choice is proven to a verifier, access is permanently revoked. Such scatch-off cards can, for example, be used to download digital media and later request a refund if it was never accessed. Again, our protocol supports similar, but more general use cases; Data can be accessed until it is transferred to another machine.

5 Conclusion

The ability to endlessly copy data poses security challenges in many settings. We presented a protocol to state-continuously transfer state of protected modules from one machine to another. We believe that this new security feature of protected-module architectures enables many use cases that were not possible before.

References

1. P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of c code executing in an unverified context. In *Accepted for publication in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, Jan. 2015.
2. P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012)*, pages 171–185, Los Alamitos, CA, USA, 2012. IEEE Computer Society.
3. I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.
4. E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell. BootJacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 555–564, New York, NY, USA, 2008. ACM.
5. J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, pages 45–60, 2008.
6. Intel Corporation. *Software Guard Extensions Programming Reference*, 2013.
7. J. King-Lacroix and A. Martin. Bottlecap: A credential manager for capability systems. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 45–54, New York, NY, USA, 2012. ACM.

8. P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, page 10. ACM, 2014.
9. R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.
10. J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'10)*, May 2010.
11. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, pages 315–328. ACM, Apr. 2008.
12. J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (Usenix'13)*. USENIX Association, Aug. 2013.
13. A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49), 1996.
14. E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Conference on Computer & communications security (CCS'13)*, 2013.
15. B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'11)*, May 2011.
16. M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. In *Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)*, 2014.
17. M. Patrignani, D. Clarke, and F. Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*, 2013.
18. R. Strackx, B. Jacobs, and F. Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Annual Computer Security Applications Conference (ACSAC'14)*, 2014.
19. R. Strackx, B. Jacobs, and F. Piessens. ICE: A passive, high-speed, state-continuity scheme (extended version). CW Reports CW672, KU Leuven, August 2014.
20. R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Computer and Communications Security (CCS'12)*, October 2012.
21. R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks (SecureComm'10)*, pages 344–361. Springer, 2010.
22. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.
23. M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, STC '07, 2007.
24. A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 430–444, Washington, DC, USA, 2013. IEEE Computer Society.